Desarrollo de un Framework 3D para GameMaker

Agustín Molina Uasuf

Resum— En este ensayo se describen los elementos del desarrollo de un Framework 3D para el motor gráfico GameMaker con el objetivo de construir un juego de estilo acción en tercera persona. Este trabajo requiere la integración en dicho motor pensado para las dos dimensiones, componentes tales como la importación de modelos 3D, las animaciones por esqueleto, la iluminación o las colisiones en un espacio tridimensional. Finalmente buscando la recreación de un viejo proyecto de videojuego con el fin de ejercer una comparación entre las dos versiones con más de seis años de diferencia entre ellas.

Paraules clau— GameMaker, Gráficos por Computador, Shaders, Animación de Esqueleto, Colisiones 3D, Motor Gráfico, Videojuegos.

Abstract— In this essay, it describes the elements of the development of a 3D Framework for the GameMaker game engine with the objective of building a third person action game. This project requires the integration in the said engine thought for two-dimension components such as 3D model importer, skeletal animations, illumination or collisions on a three-dimensional space. Looking for remaking an old proyect of a video game with the purpose of establishing a comparison between the two versions with six years of difference between them.

Index Terms— GameMaker, Computer Graphics, Shaders, Skeletal Animation, 3D Collisions, Game Engine, Videogames.

1 Introducción

Este trabajo comienza en 2015 con un proyecto de fin de bachillerato (*Treball de Recerca*), donde junto a otros dos compañeros se planteó el objetivo de desarrollar el prototipo de un videojuego. Bajo la responsabilidad como programador en este Trabajo ocurre una primera aproximación hacia la programación, la cual si bien podría considerarse funcional, se encontró limitada por la condición de aprendiz.

Tras cinco años de carrera y bajo el pretexto de este TFG, se decide retomar ese mismo proyecto y rehacer, utilizando las mismas herramientas, el videojuego que se programó en bachillerato, aplicando en el camino los conocimientos que se han adquirido gracias al tiempo y el haber cursado el grado en ingeniería informática.

El trabajo original constaba de una demo jugable con cuatro habitaciones, dos enemigos, cinco objetos y un jefe de nivel. A nivel jugable se nos mostraba una vista cenital y un personaje que era capaz de desplazarse y atacar. El objetivo de dicha versión de prueba era abrirse paso a través de las habitaciones derrotando a los enemigos y obteniendo por un lado experiencia para subir de nivel y por ende que las estadísticas del jugador mejoren, y por el otro dinero para comprar objetos en la tienda. En la habitación final el jugador se encuentra con un enemigo el cual requeriría de un mayor nivel de habilidad.

El proyecto en el que se basa este TFG fue programado utilizando GameMaker Studio [1], un motor gráfico especializado en la creación de juegos 2D (e.g. *Hotline Miami [2], Undertale [3], Hyper Light Drifter [4]*). Este motor utiliza un lenguaje propio de scripting llamado *GameMaker Language [5]* (*GML*) de gramática similar a Javascript. Si bien con GMS se pueden realizar juegos en 3D, las funcionalidades que este ofrece son de bajo nivel y por lo tanto poco accesibles para el perfil de principiante.

Dadas estas condiciones, la propuesta para este trabajo es la de, utilizando los conocimientos adquiridos tanto en programación como en *game design*, rehacer esta demo jugable con este mismo motor, buscando un acabado acorde a la profesionalidad requerida para un juego que se quiera comercializar. Como añadido se busca trasladar la idea inicial de juego de las 2D a las 3D.

Como se ha nombrado anteriormente, GameMaker Studio es un motor gráfico especializado en las 2D por lo que carece de muchas herramientas imprescindibles para el desarrollo de un juego en tres dimensiones. Dado esto, como objetivo secundario, la mitad del proyecto estará enfocado en el desarrollo de un framework que cuente con estos elementos necesarios tales como:

- Cámara con proyección en perspectiva 3D.
- Importación de modelos.
- Rigging y animación.
- Editor de escenarios.
- Efectos visuales (partículas, sombras, luces).

- *E-mail de contacto: Molina.Uasuf@gmail.com*
- *Mención realizada: Computación*
- *Trabajo tutorizado por: Jorge Bernal (Ciencias de la Computación)*
- *Curs 2020/21*

Dicho framework se publica de manera abierta y gratuita para la comunidad de GameMaker pues diversos elementos estarán inspirados en librerías ya implementadas por usuarios del motor.
Finalmente establecer como otro objetivo secundario que se busca añadir en lo jugable en el proyecto son elementos de *Inteligencia Artificial* en los enemigos que pueda garantizar una buena experiencia para el jugador.

2 ESTADO DEL ARTE

Hoy en día el desarrollo de videojuegos se ha visto democratizado dada la amplia oferta de motores gráficos que encontramos en el mercado tanto gratuitos como de pago. Motores como Unity [6] o Unreal [7] han abierto las puertas de la programación a miles de usuarios no especializados en la materia que, por otras vías, si hubiesen querido realizar el mismo proyecto habrían requerido de unos mayores conocimientos de informática.
Y así es como nos encontramos con que los motores gráficos son excelentes herramientas que encapsulan prácticamente todas las funcionalidades que un desarrollador de videojuegos pueda necesitar. Con esto nos referimos a elementos tan troncales como el *Game loop* (refiérase al esqueleto esencial del motor o la secuencia de eventos donde se ordenará el código de nuestro juego) o el manejo de gráficos ya sean 2D o 3D. Motores más complejos como los que conocemos hoy en día cuentan con editores de escenario, manejo de audio y de input, funcionalidades de *networking* e inclusive acceso a una gran comunidad donde poder encontrarse con recursos e información ofrecidos por otros usuarios del motor.

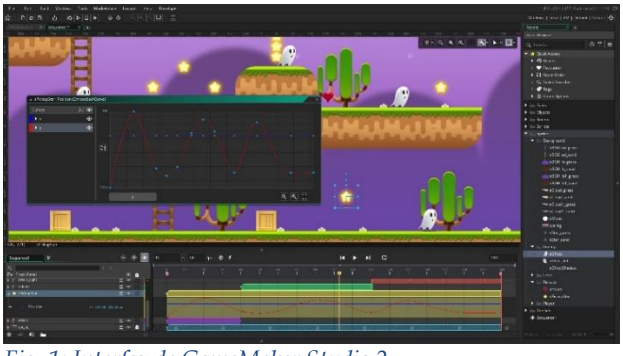

Fig. 1: Interfaz de GameMaker Studio 2

En estos tiempos el mercado de motores gráficos está principalmente liderado [8] por Unity y Unreal Engine. A estos les siguen otros como Godot [9], RPG Maker [10], Cry Engine [11] y GameMaker. Tal y como observamos la oferta es amplia y por ello también las características de estos. Algunos son de pago, otros son gratuitos, otros requieren de un porcentaje de las ventas y otros son totalmente open source [12].
Entre estos motores la elección de emplear en este TFG GameMaker se ve influenciada principalmente por la experiencia previa con el programa y en menor medida por el atractivo de proponer un proyecto que se encuentra en un punto intermedio entre hacer un juego con un motor gráfico desde cero o utilizar uno con plenas funcionalidades en el ámbito 3D. Como añadido la utilización de GameMaker frente a otros motores como Unity o Unreal ofrece un mayor nivel de control sobre la gestión del renderizado debido a que trabajamos desde un nivel más bajo.

3 REQUISITOS

3.1 Requisitos hardware

- **Requisitos mínimos [13] (relacionados con GameMaker)**
  - 64bit Intel Dual Core CPU.
  - 2GB RAM.
  - Tarjeta gráfica compatible con DX11.
  - Microsoft 64bit Windows 7.
  - 3GB de espacio libre en el disco duro.
- **Requisitos recomendados**
  - Intel Core i5-8250U.
  - 8 GB DDR4 RAM.
  - NVIDIA GeForce MX150.
  - Microsoft 64bit Windows 10.
  - 256GB SSD.

3.2 Requisitos software

GameMaker Studio 2 es un motor gráfico de pago y por lo tanto los proyectos realizados en él requieren de una licencia válida para poder ser editados. En la actualidad contamos con dos tipos de modalidades licencias: pago único por 91€ y pago anual por 36€ [14].
GameMaker Studio es capaz de exportar a diversas plataformas ya sean ordenadores, móviles o consolas. Sin embargo, las licencias bajo las que trabajamos permiten la exportación en Windows, MacOS, Ubuntu y HTML5 por lo que para poder ejecutar el proyecto el usuario requerirá de alguna de las plataformas mencionadas anteriormente.

4 RIESGOS DEL PROYECTO

- **Riesgo:** No poder implementar en el framework animaciones 3D por medio de esqueletos.
  Probabilidad: Alta
  Impacto: Bajo.
  Contingencia: El juego utilizará modelos estáticos sin animación o en su defecto animaciones por vértices.

- **Riesgo:** No ser capaz de adquirir los conocimientos necesarios de animación y modelaje 3D.
  Probabilidad: Alta
  Impacto: Medio.
  Contingencia: Puedo recurrir a terceros para que realicen labores artísticas del juego.

- **Riesgo:** Mala planificación del tiempo, lo cual desencadenaría en no cumplir los objetivos jugables del proyecto o los estándares de calidad deseados.
  Probabilidad: Bajo
  Impacto: Alto.
  Contingencia: Simplificar o prescindir de elementos menos esenciales del proyecto tales como la IA o los efectos visuales.

- **Riesgo:** Incapacidad de implementar las funcionalidades básicas de 3D para el juego.
  Probabilidad: Bajo
  Impacto: Alto.
  Contingencia: Acceder a las librerías ya existentes que hacen lo que yo deseo fabricar desde cero.

5 METODOLOGÍA

Para el source control utilizado en este proyecto se han utilizado las herramientas GIT [15] y Github [16].
En cuanto a la organización del proyecto junto al tutor se han realizado reuniones semanales para comprobar el estado del proyecto. Para trabajos relativos al arte, de manera auxiliar se contrató por comisión a un modelador 3D.

6 PLANIFICACIÓN

Este proyecto se extiende en cuatro meses y consta de dos partes principales las cuales son:

- **Desarrollo del framework 3D**. De una duración estimada de tres meses, el objetivo será desarrollar todos los componentes necesarios para la realización de un juego en tres dimensiones. Los componentes planificados son una cámara de proyección en perspectiva, importación de modelos, animación por esqueleto y colisiones de volúmenes.
- **Implementación de una demo jugable**. Con una duración estimada de un mes, aquí se busca utilizar las herramientas construidas para reconstruir la versión del juego realizado seis años atrás. Esta demo debe contener el control de un personaje principal (desplazamiento, habilidades, estadísticas), al menos un enemigo con una máquina de estados a modo de inteligencia artificial

Como se puede apreciar el desarrollo del framework comprende el 75% del trabajo mientras que la demo jugable supone el 25% restante.

7 DESARROLLO DEL FRAMEWORK

En este apartado se explicarán todos los procesos de desarrollo del proyecto y los componentes que lo conforman.

7.1 Librería matemática

Una framework preparado para trabajar en tres dimensiones requiere de un gran volumen de matemáticas de las cuales GameMaker carece. Por lo tanto, para este proyecto se ha desarrollado un conjunto de librerías con las que tratamos los siguientes conceptos matemáticos:

- **Vectores** (Vector2, Vector3 y Vector4) para codificar posición, translación, dirección, colores, etc.
- **Quaterniones** (Quaternion [17] y DualQuaternion [18]) que nos sirven para almacenar rotaciones y en el caso de dual cuaternion una rotación más una translación.
- **Matrices** (Matrix3x3 y Matrix4x4) donde tenemos las funcionalidades básicas de las matrices como la multiplicación, determinante, inversa y otros elementos necesarios para codificar las transformaciones en el espacio.

7.2 Cámara en perspectiva

Como se ha nombrado anteriormente GameMaker es un motor gráfico pensado principalmente para la realización de juegos en dos dimensiones. Tras un primer vistazo podemos ver que este motor simplemente es capaz del dibujado de imágenes sobre un plano bidimensional. Sin embargo, mirando con más detenimiento podemos observar que GameMaker realmente funciona de manera interna en **tres dimensiones**, eso significa que de manera predeterminada este cuenta con una cámara de proyección ortográfica y el dibujado de polígonos paralelos al plano X e Y.
De esta manera GameMaker nos ofrece una serie de funcionalidades de bajo nivel para modificar el funcionamiento intrínseco de la cámara. Para ello accederemos y modificaremos las matrices de Posición y de Proyección.

- **Matriz de Posición** conocida comúnmente como LookAt Matrix [19]. Describe tres vectores que son **position**, **front** y **up**. El primero describe la posición en el espacio de la cámara. El segundo vector indica hacia qué dirección mira la cámara. Finalmente, el vector up indica cuál es la inclinación de la cámara. De manera predeterminada GameMaker mantiene invariable el vector front en [0,0,1] lo que es igual a mirar en dirección al eje Z o eje de profundidad en las dos dimensiones. De esta manera adquirimos la capacidad en cualquier punto del espacio una camara que apunte en cualquier dirección deseada.

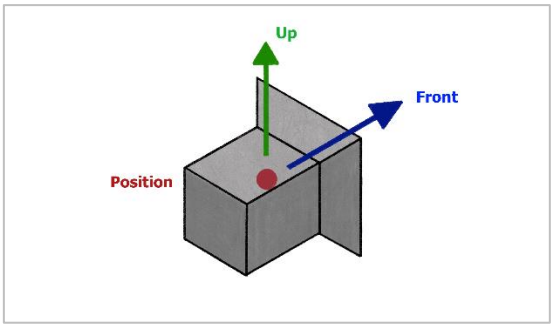

Fig. 2: Vectores que conforman la matriz de proyección

- **Matriz de Proyección**. Esta matriz determina de qué manera se observa el mundo. Principalmente en los gráficos por computador se utilizan el modo ortográfico (el predeterminado para game maker) y el modo perspectiva. Este último es el que debemos utilizar para adquirir la sensación de profundidad.

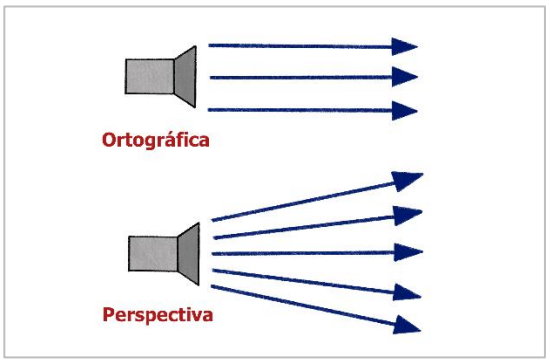

Fig. 3: Proyecciones ortográfica y de perspectiva

Una vez creadas nuestras matrices lo que queda es aplicarlas a la cámara activa para que en la *Rendering Pipeline* muestre de la manera deseada el mundo que estamos creando.

7.3 Importación de modelos

En este apartado sobre la importación de modelos debemos explicar el tratamiento de la geometría en los gráficos por computador, pues en la mayoría de las aplicaciones gráficas, el ordenador trabaja con triángulos. Y GameMaker no es una excepción. Si bien este motor trabaja de manera determinada con Sprites (imágenes planas), estas esencialmente son un rectángulo conformado por dos triángulos unidos en la hipotenusa.

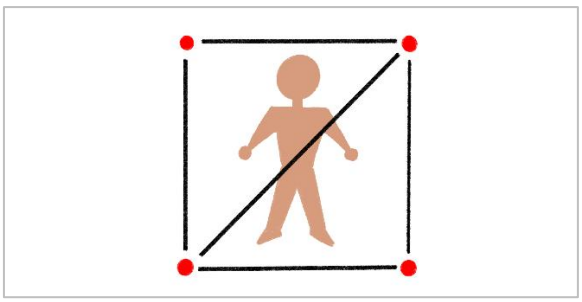

Fig. 4: Polígonos que conforman un Sprite

Sin embargo, para nuestro proyecto nosotros lo que necesitamos no son cuadrados con imágenes inscritas sino modelos 3D con volumen que representen nuestros personajes en el juego.

Para pasar de simples planos a volúmenes GameMaker nos permite directamente enviarle a la gráfica nuestros propios polígonos sin necesidad de que estos formen un rectángulo. Y en su totalidad, los modelos en 3D no dejan de ser un conjunto de triángulos situados en el espacio.

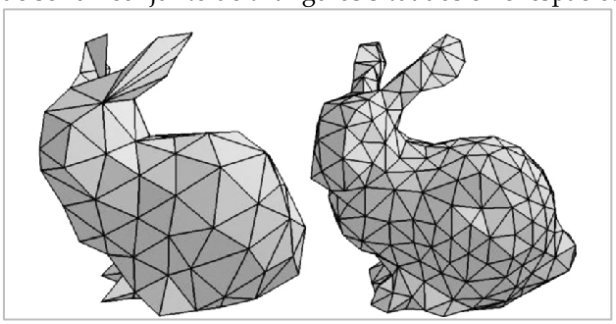

Fig. 5: Modelo conformado por varios triángulos

Para el diseño de los modelos en este proyecto hemos decidido utilizar Blender [20], el cual es una herramienta de código abierto especializada en esta tarea. Además, Blender ofrece diversos formatos de exportación de modelos, entre ellos .OBJ [21] y .DAE [22] los cuales son los que hemos implementado en este proyecto. Por un lado .OBJ permite almacenar exclusivamente la geometría del objeto mientras que en .DAE también se guardan el esqueleto y las animaciones. Para ello nuestros importadores han de leer tres características principales por cada vértice del triángulo. Estas son la posición, la normal y las coordenadas de las texturas.

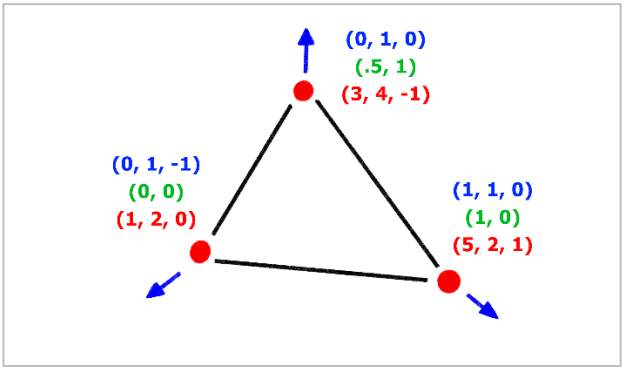

Fig. 6: Datos contenidos por cada vértice de un triángulo

Observando la figura anterior, en rojo podemos observar la posición de cada vértice. En azul se nos indica la normal a estos vértices que en esencia es la dirección a la que el vértice apunta (necesario para la iluminación). El verde son las coordenadas de las texturas.

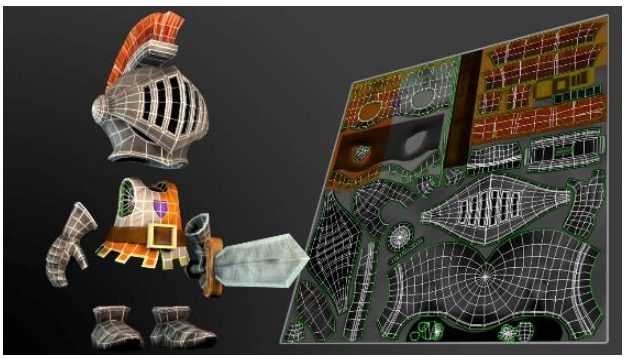

Fig. 7: Textura 2D mapeada en un modelo 3D utilizando UVs

Por lo tanto, de los archivos de texto donde se guardan los modelos almacenaremos estos modelos a modo de buffer de datos que es lo que enviaremos a la gráfica.

```
1  2  0  0  1 -1  0  0
5  2  1  1  1  0  1  0
3  4  1  0  1  0  0 .5
```

En los modelos contemplados para la animación por esqueletos a cada vértice le añadimos dos campos de tres elementos cada uno los cuales son los índices de las articulaciones que influencian el vértice y el porcentaje que cada una de esta lo afecta, quedando como resultado una cadena de el siguiente estilo para cada triángulo.

```
 1  2  0  0  1 -1  0  0  1  2  3 .6 .2 .2
 5  2  1  1  1  0  1  0  2  1  4 .7 .2 .1
 3  4  1  0  1  0  0 .5  1  2  4 .8 .1 .1
```

7.4 Animación por esqueleto

Este componente es el que se encargará de ejercer transformaciones individuales a cada vértice de la malla con el objetivo de poder animar el modelo (andar, correr, saltar, golpear). Para realizar esto definimos dos subcomponentes principales los cuales son el esqueleto y la animación.

7.4.1 Esqueleto y sus articulaciones

El esqueleto del modelo se ve representado internamente como una jerarquía de nodos llamados articulaciones (joints) mientras que las líneas unen a estas son los huesos. Cada una de estas articulaciones está compuesta por una rotación, una traslación y una lista con todos los hijos de este nodo. La rotación y la traslación de cada nodo se codifica en forma de cuaternión dual, el cual es un elemento matemático similar a las matrices 4x4 de transformación, pero a cambio de perder información sobre la transformación de escalado ganamos en espacio (pues un cuaternión dual ocupa ocho valores reales mientras que la matriz ocupa dieciséis).

```
Spine
|   Spine2
|   |   Spine3
|   |   |   Neck
|   |   |   |   Head
|   |   |   Shoulder_L
|   |   |   |   Arm1_L
|   |   |   |   |   Arm2_L
|   |   |   |   |   |   Hand_L
|   |   |   Shoulder_R
|   |   |   |   Arm1_R
|   |   |   |   |   Arm2_R
|   |   |   |   |   |   Hand_R
|   Leg1_L
|   |   Leg2_L
|   Leg1_R
|   |   Leg2_R
```

Fig. 8 Jerarquía de articulaciones de un esqueleto

En la función Update() del esqueleto se ejecuta en forma de cascada comenzando desde el nodo raíz. Cada articulación busca como objetivo obtener un cuaternión dual llamado deltaWorld. La operación para obtener este elemento deberemos realizar la siguiente cadena de operaciones:

```
currentLocal = localDQ * keyframe[i];

if ( parent != NULL )
    currentWorld = parent.currentWorld * currentLocal;
else
    currentWorld = currentLocal;

deltaWorld = currentWorld * inverse(worldDQ);
```

Este proceso se basa en tres pasos:

- En primer lugar, calculamos nuestro currentLocal. Este cuaternión dual es el resultado de multiplicar nuestro cuaternión dual base (el que tenemos en estado de reposo) por el cuaternión dual del fotograma actual de la animación.
- En segundo lugar, calculamos currentWorld. Para el nodo raíz el espacio local y el espacio mundo es el mismo, pero para el resto de los nodos la transformación en coordenadas mundo serán el resultado de multiplicar las coordenadas mundo de nuestro padre por las locales nuestras.
- En último lugar, calculamos delta, el cual es el resultado de multiplicar el currentWorld por el inverso de el world base. Esto nos dará un cuaternión dual que supondrá un diferencial delta del estado original de nuestra transformación.

Para poner un ejemplo sobre el funcionamiento de este sistema, en caso de que la rotación del hueso en este fotograma sea nula respecto al estado base, si seguimos las operaciones veremos que terminamos multiplicando currentWorld por el inverso del world base, que, en este caso, siendo los dos el mismo cuaternion dual estaríamos multiplicando uno por el inverso de si mismo, obteniendo el cuaternión identidad como resultado. Esto haría que los vértices del modelo asociados a esa articulación no se viesen afectados.

7.4.2 Sistema de animación

Junto al esqueleto, este framework cuenta con un sistema de animación que se encarga almacenar todas las alteraciones al esqueleto para un fotograma específico del clip animado.

```
animations = {
    "idle"  : [ DQ_0, DQ_1, ..., DQ_n],
    "run"   : [ DQ_0, DQ_1, ..., DQ_n],
    "run"   : [ DQ_0, DQ_1, ..., DQ_n]
};
```

En nuestro proyecto este sistema se integra dentro de la clase articulación donde tenemos un diccionario donde las llaves son el nombre de la animación (*jump, run, hit*) y el contenido es una lista donde el indice es el número del fotograma y el valor un cuaternión dual con el diferencial del estado base de la articulación hacia el actual.

7.5 Colisiones 3D

Las colisiones en un videojuego son un elemento clave para implementar sistemas de batalla donde nuestro personaje ha de golpear al otro. Para este objetivo GameMaker cuenta con una serie de funciones para detectar colisiones respecto a cuerpos bidimensionales (círculos, rectángulos, líneas). Para este framework requerimos que nuestro sistema detecte colisiones de figuras geométricas situadas en un espacio 3D (cubos, esferas, líneas con coordenadas en tres dimensiones).

En este framework contemplamos las colisiones de las siguientes figuras:

- **Puntos**: Definidos únicamente por una coordenada en el espacio.
- **Esferas**: Definidas por su posición y el radio.
- **Cilindros**: Definidos por posición, radio y altura.
- **Lineas**: Definidas por dos coordenadas en el espacio.

El proceso de comprobar las colisiones entre dos cuerpos

dependerá del tipo de estos dos, por ello hemos de realizar una tabla de colisiones donde cada celda es una función distinta.

	Línea	**Cilindro**	**Esfera**	**Linea**
Punto	col_line_point	col_cilinder_point	col_sphere_point	col_line_point
Esfera	col_line_sphere	col_cilinder_sphere	col_sphere_sphere	
Cilindro	col_line_cilinder	col_cilinder_cilinder		
Línea	col_line_line			

Fig. 9: Matriz de colisión

En la figura anterior podemos ver la matriz de colisión donde cada eje es una de las dos figuras las cuales estamos comprobando su colisión y el contenido de las celdas es la función a la que tenemos que llamar. En cada una de estas funciones ejerceremos unos procesos matemáticos distintos dependiendo de las figuras que llamamos.

Una vez ya existen las funciones de colisión para hacerlas funcionar, cada uno de nuestros objetos del juego que queramos que comprueben colisiones tendrán que instanciar individualmente un colisionador, el cual es una estructura la cual almacena el tipo de figura que representa a nuestro personaje y la función Check(), a la cual como parámetro le pasaremos otro objeto. Dentro de esta función Check() haremos un switch case para redirigirnos, dependiendo del tipo de figura del otro objeto, a la función adecuada.

8 DESARROLLO DE LA DEMO

Una vez completado el framework comienza el desarrollo de la demo, la cual destacamos cuatro elementos troncales respecto a su elaboración: El diseño, el personaje, la cámara y la IA.

8.1 Diseño

Este apartado de diseño engloba los conceptos de diseño de mecánicas y diseño artístico.

Por diseño de mecánicas este juego cae en el grupo de los llamados *Souls-Like [23]* los cuales son los juegos influenciados por la saga *Dark Souls*. Estos juegos cuentan como características principales:

- Un combate cuerpo a cuerpo complejo donde cada enfrentamiento podría suponer el fracaso del jugador (al contrario de la norma donde el avatar es capaz de enfrentarse a varios enemigos al mismo tiempo sin muchos problemas)
- Una penalización severa por el fracaso haciéndole perder al jugador parte de la experiencia ganada en caso de no ser capaz de retornar al último lugar donde fue asesinado.
- Un diseño de niveles donde se premia la exploración del mapa ofreciendo la posibilidad de desbloquear atajos con tal de hacer la navegación del escenario más rápida.

En cuanto al diseño artístico del juego se cuenta con escenarios situados en el mar mediterráneo y con la iconografía de las culturas que se encuentran cerca de este (España, Grecia, Marruecos).
Para realizar los modelos 3D se utilizó la herramienta Blender, el cual es un software de código abierto especializado en el modelado y la animación.
Respecto al diseño del escenario nos basamos en el diseño original de una isla separada en seis escenarios distintos la cual fue traída al 3D.

Fig. 10: Versión original del mapa del juego

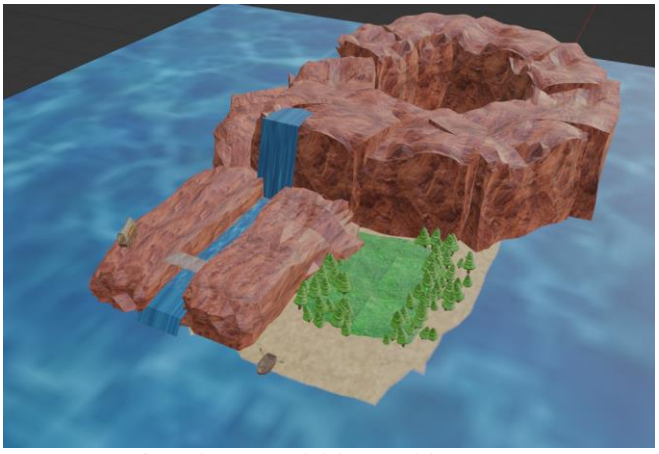

Fig. 11: Versión tridimensional del mapa del juego

Para el diseño del personaje se contrató a un modelador 3D por comisión el cual para comenzar a trabajar requirió de un concept art [24] (ilustración que busca dar una idea principal sobre un diseño).

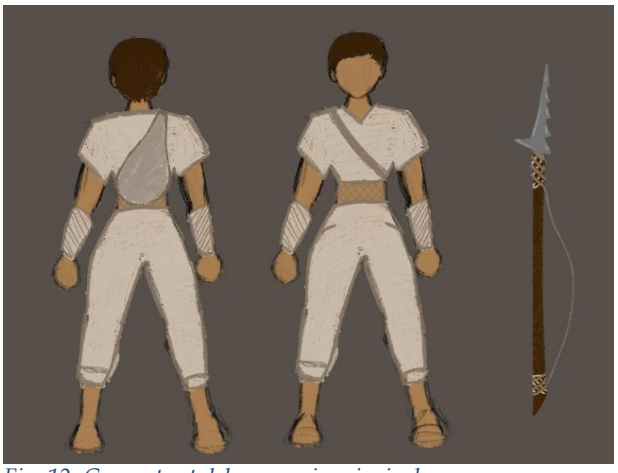

Fig. 12: Concept art del personaje principal

8.2 Personaje y Estadísticas

Tanto el personaje principal como los enemigos controlados por la máquina contienen una Máquina de Estados Finitos [25] con el objetivo de organizar su control y encapsular la programación de funcionalidades de los objetos. En el caso del personaje esta máquina tiene los estados de Idle, Move y Attack. Cada estado de la máquina está asociado a una animación distinta del personaje las cual se iniciará automáticamente tras entrar a un nuevo estado.

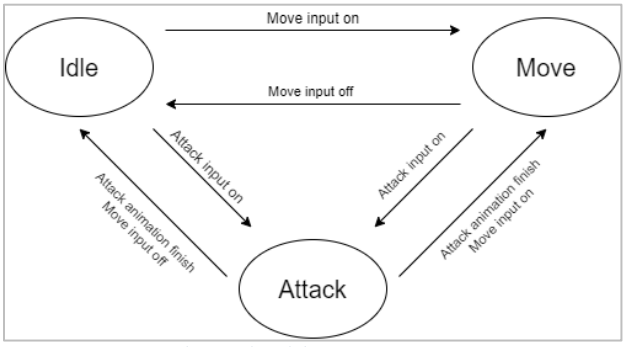

Fig. 13: Maquina de estados del personaje

8.3 Cámara

Nuestro juego cuenta con una cámara de perspectiva en tercera persona, lo que significa que la cámara se sitúa a una distancia determinada enfocando hacia el personaje.

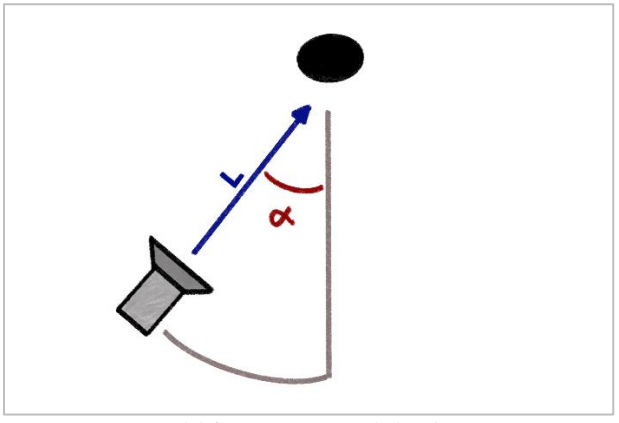

Fig. 14: Esquema del funcionamiento de la cámara

Nuestra cámara dada una distancia L y un angulo Alfa de hacia a donde mira el personaje la posición de la cámara se definiría según la siguiente ecuación:

```
Camera.x = Player.x + L * cos(Alpha);
Camera.y = Player.y + L * sin(Alpha);
```

Sin embargo, esto resultaría muy brusco para el jugador y podría dar lugar al mareo, por lo que al movimiento de la cámara le aportamos un factor de suavizado utilizando la función de interpolación lineal [26].

```
function Lerp(a, b, amount)
    return a + (b - a) * amount;

Camera.x = Lerp(Camera.x, Player.x + L * cos(Alpha) );
Camera.y = Lerp(Camera.x, Player.y + L * sin(Alpha) );
```

Esta función aproxima un valor A hacia el valor B en una proporción Amount de la distancia que los separa haciendo que a cada actualización del juego la cámara se acerque en el porcentaje Amount hacia la posición deseada.

8.4 IA Enemigos

El juego cuenta con dos tipos de enemigos: Uno que persigue al jugador y otro que huye de él.
Estos enemigos al igual que el personaje principal cuenta con una máquina de estados con Idle, Move y Attack.

En el caso del enemigo que persigue al jugador definimos dos rangos. Si el jugador se encuentra a una distancia inferior al primer rango, el enemigo perseguirá al personaje. En caso de superar el segundo rango el enemigo ejecutará su animación de ataque

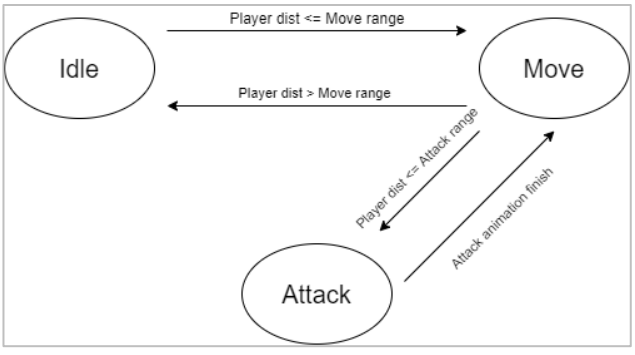

Fig. 15: Máquina de estados del enemigo que persigue

El segundo enemigo a diferencia del primero buscará huir del personaje para dispararle a distancia en caso de que se esté a una distancia superior al rango designado.

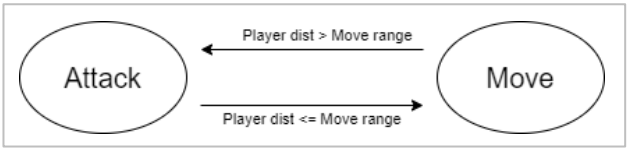

Fig. 16: Máquina de estados del enemigo que huye

8 Resultados

Como resultado de este proyecto los objetivos propuestos respecto al desarrollo del framework 3D han sido cumplidos exitosamente. Los componentes planeados se pudieron implementar y estos funcionan de una manera relativamente optima pues permite ejecutar el juego a tiempo real.

En cuanto a la realización de la demo, el desarrollo del componente para animaciones se prolongó impidiendo que se implementaran todos los elementos deseados en la demo.

9 Conclusiones

Este proyecto ha sido un reto interesante para expandir los conocimientos relativos a los gráficos por computador y en elementos matemáticos aprendidos como los cuaterniones duales.
Las animaciones por esqueleto han resultado el mayor desafío de este desarrollo, pero a su vez eran en gran parte la motivación para realizar este proyecto.
Este framework se ofrecerá de manera totalmente gratuita a la comunidad de GameMaker debido al gran apoyo recibido por parte de la comunidad, con la esperanza que la gente acceda a él y lo expanda a sus necesidadaes.

Agradecimientos

Me gustaría agradecer en primer lugar a Jorge Bernal por la confianza depositada en el proyecto. Además, me gustaría agradecer a la comunidad de GameMaker y en particular al usuario TheSnidr por su ayuda en la implementación de las animaciones por esqueleto. Finalmente agradecer a Crimsongcat por los modelos aportados al proyecto.

Bibliografía

- Venuta, A. (2014). Implementing Skeletal Animation. https://veeenu.github.io/blog/implementing-skeletal-animation/
- Collada Tutorial. http://wazim.com/collada-tutorial1/
- Graphics for Games. Newcastle University. https://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/
- OpenGL Tutorial: Rotations. http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/
- GameMaker Studio 2 Documentation. https://manual.yoyogames.com/

Referencias

[1] GameMaker Studio 2. https://www.yoyogames.com/es
[2] Hotline Miami. https://hotlinemiami.com/
[3] Undertale. https://undertale.com/
[4] Hyper Light Drifter. https://heartmachine.com/hyper-light
[5] GameMaker Language. https://docs2.yoyogames.com/source/_build/3_scripting/3_gml_overview/index.html
[6] Unity. https://unity.com/es
[7] Unreal Engine. https://www.unrealengine.com/en-US/
[8] Toftedahl, M. (2019). Which are the most commonly used Game Engines? https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php
[9] Godot Engine. https://godotengine.org/
[10] RPG Maker. https://www.rpgmakerweb.com/
[11] Cry Engine. https://www.cryengine.com/
[12] Open source iniciative. https://opensource.org/
[13] Requisitos mínimos de GameMaker Studio 2. https://docs2.yoyogames.com/
[14] Licencias de GameMaker Studio 2. https://www.yoyogames.com/get
[15] Git. https://git-scm.com/
[16] Github. https://github.com/
[17] Quaternion. http://www.chrobotics.com/library/understanding-quaternions
[18] Dual Quaternion. https://www.euclideanspace.com/maths/algebra/realNormedAlgebra/other/dualQuaternion/index.htm
[19] LookAt Matrix. https://www.geertarien.com/blog/2017/07/30/breakdown-of-the-lookAt-function-in-OpenGL/
[20] Blender. https://www.blender.org/
[21] Wavefront OBJ File. https://www.cs.cmu.edu/~mbz/personal/graphics/obj.html
[22] Collada DAE File. https://www.khronos.org/collada/
[23] Albano, A. (2020). ¿Qué es realmente un Soulslike? https://pressover.news/articulos/que-es-realmente-un-soulslike/
[24] Fitzgerald, R. (2019) What is Concept Art? https://www.cgspectrum.com/blog/what-is-concept-art
[25] Álvarez, R. Introducción a las Máquinas de Estado Finito. http://tecbolivia.com/index.php/articulos-y-tutoriales-microcontroladores/13-introduccion-a-las-maquinas-de-estado-finito
[26] Interpolación Lineal. https://matematica.laguia2000.com/general/interpolacion-lineal